1   **Exhibit E**
2
3   **FROM MEDIA.CPP**
4
5   ```
//+--
```
6    `//`
7    `// Function:   CWMPMedia::FetchAlbumArtURLForMedia`
8    `//`
9    `//------------------------------------------------------------------------`
10   `HRESULT`
11   `CWMPMedia::FetchAlbumArtURLForMedia( MediaReservedItemId mriid, VARIANT* pvtitem )`
12   `{`
13   `    HRESULT        hr = S_OK;`
14   `    WBSTRString    wstrMediaFilepath;`
15   `    CURL           curlMediaFilename;`
16   `    VARIANT_BOOL   vtIsAvailable = VARIANT_FALSE;`
17
18   `    hr = IsAvailable( &vtIsAvailable );`
19
20   `    if( SUCCEEDED( hr ) &&`
21   `        vtIsAvailable )`
22   `    {`
23   `        if( IsWhistlerOrBetter() )`
24   `        {`
25   `            hr = get_sourceURL( &wstrMediaFilepath );`
26
27   `            if( S_OK == hr )`
28   `            {`
29   `                hr = curlMediaFilename.Set( UI_PATH, wstrMediaFilepath );`
30   `            }`
31   `            if( SUCCEEDED( hr ) )`
32   `            {`
33   `                hr = curlMediaFilename.SetFileName( NULL );`
34   `            }`
35   `            if( SUCCEEDED( hr )  )`
36   `            {`
37   `                hr = curlMediaFilename.Get( SF_FILEIO, wstrMediaFilepath );`
38   `            }`
39   `            if( SUCCEEDED( hr ) )`
40   `            {`
41   `                if (!CURLHelper::ShareKnownToBeUp(curlMediaFilename))`
42   `                {`
43   `                    hr = HRESULT_FROM_WIN32(ERROR_BAD_NETPATH);`
44   `                }`
45   `                else if (!GetCustomAlbumArt( wstrMediaFilepath, pvtitem ))`
46   `                {`
47   `                    WString wszCollectionFilename;`
48   `                    CComVariant var;`

```
49
50              (void)CWMPItemDataMgr::GetAttributeByAtom(
51     ITEMDATA_GETATTRIBUTE_MINIMALMETADATA, CSchemaMap::kiIndex_WMWMCollectionID, NULL, 0,
52     &var );
53              if ((var.vt == VT_BSTR) && (var.bstrVal))
54              {
55                  (void)wszCollectionFilename.Sprintf(   L"%s%s%s",   g_kwszAlbumArtPrefix,
56                                          var.bstrVal,
57                                          ((kmriidSmallAlbumArtURL == mriid) ? g_kwszArtSuffixSmall :
58     g_kwszArtSuffixLarge)   );
59              }
60
61              const WCHAR * rgURLs[2];
62              rgURLs[0]   = wszCollectionFilename;
63              rgURLs[1]   = ((kmriidSmallAlbumArtURL == mriid) ?
64     g_wszWMPL_ALBUM_ART_SMALL_FILENAME : g_wszWMPL_ALBUM_ART_LARGE_FILENAME);
65
66              for (long nIndex = 0; nIndex < RGSIZE(rgURLs); nIndex++)
67              {
68                  if (!rgURLs[nIndex])
69                  {
70                      continue;
71                  }
72
73                  CURL urlTest;
74
75                  hr = curlMediaFilename.CopyTo( urlTest );
76                  if (SUCCEEDED(hr))
77                  {
78                      hr = urlTest.PathAppend( rgURLs[nIndex] );
79                  }
80                  if( SUCCEEDED( hr ) )
81                  {
82                      hr = CURLHelper::VerifyFileExists(urlTest);
83                  }
84                  if( SUCCEEDED( hr ) )
85                  {
86                      hr = urlTest.Get( SF_FILEIO, wstrMediaFilepath );
87                  }
88                  if( SUCCEEDED( hr ) )
89                  {
90                      pvtitem->bstrVal = WString::SysAllocString( wstrMediaFilepath );
91                      if ( NULL == pvtitem->bstrVal )
92                      {
93                          hr = E_OUTOFMEMORY;
94                      }
95                      else
96                      {
```

```
97                    pvtitem->vt = VT_BSTR;
98                  }
99                }
100               if (SUCCEEDED(hr))
101               {
102                 //
103                 // OK, if we're trying to hand back the legacy "folder.jpg" calls,
104                 // then we're going to attempt to ensure that we don't hand it back
105                 // incorrectly.
106                 //
107
108                 if (nIndex == 1)   // (second entry in our array -- folder.jpg)
109                 {
110                    if (!ShouldWeUseHelixArt( curlMediaFilename ))
111                    {
112                       ::VariantClear( pvtitem );
113                       hr = HRESULT_FROM_WIN32(ERROR_FILE_NOT_FOUND);
114                    }
115                 }
116
117                 break;
118               }
119             }
120           }
121         }
122       else
123       {
124          hr = E_UNEXPECTED;
125       }
126     }
127     else
128     {
129        // Get TOC & Album Art URL from Library?????
130        hr = E_INVALIDARG;
131     }
132
133     //
134     // If we couldn't get the customized album art, then call through the item data manager
135     // to see if we can get the attribute from a library media (if this is one)
136     // and just return the URL to download the alubm art
137     //
138
139     if( FAILED( hr ) )
140     {
141        if ( kmriidSmallAlbumArtURL == mriid )
142        {
143           hr = CWMPItemDataMgr::GetAttributeByAtom( (ITEMDATA_GETATTRIBUTE_INTERNALDATA |
144     ITEMDATA_GETATTRIBUTE_MINIMALMETADATA),
```

```
145                                    ITEMDATA_GETATTRIBUTE_LIBRARY_SMALLALBUMARTYURL,
146                                    NULL,
147                                    0,
148                                    pvtitem );
149            }
150         else
151         {
152            hr = CWMPItemDataMgr::GetAttributeByAtom( (ITEMDATA_GETATTRIBUTE_INTERNALDATA |
153  ITEMDATA_GETATTRIBUTE_MINIMALMETADATA),
154                                    ITEMDATA_GETATTRIBUTE_LIBRARY_LARGEALBUMARTYURL,
155                                    NULL,
156                                    0,
157                                    pvtitem );
158            }
159        }
160     }
161
162     return ( hr );
163  }
164
165  //****************************************************************************/
166
167  bool CWMPMedia::ShouldWeUseHelixArt( const CURL& urlFolder )
168  {
169     //
170     // Problem Description: (Bug#104535)
171     //
172     // Helix delivered "folder.jpg" and "albumartsmall.jpg".  We deliver these, but also
173     // deliver "albumart_{GUID}_XXX.jpg.  This handles multiple pieces of content with the
174     // same album.  Now, since we always write the old files, then effectively "last writer"
175     // wins.
176     //
177     // Now, if you play a piece of content out of the folder that we didn't have a match
178     // for, then the album art in "folder.jpg" shouldn't be used for that piece of media.
179     //
180     // There are a couple of exceptions, though.
181     //
182     // 1)  If corona has not yet aquired metadata for an item.
183     //         * once we have metadata, we know whether or not to use the Helix "folder.jpg"
184     //         If we haven't tried to get metadata, we have to fallback to (2).
185     //
186     //         Update (9/18/2002) - We found that users tended to put folder.jpg in the file
187     //                      when we didn't match, so we've removed the check against
188     //                      metadata provider request state.
189     //
190     // 2)  Since we don't immediately load the library, we don't know (1) all the time
191     //         (primarily in the double-click case.  So, we come up with a simple compromise
192     //         in this case.  We simply need to know if "Corona" has written a piece of art
```

```
193   //          (or updated desktop.ini).  If this is true, then we shouldn't use that piece
194   //          of metadata because it was for another track (or we would have already found
195   //          the {GUID} art).  We can detect this by pulling the Buy Now URL out of the
196   //          desktop.ini file and checking the version parameter of the URL to see if
197   //          Corona wrote it.  If so, then don't use it as its almost definitely wrong.
198   //
199
200   //
201   // OK, so if we weren't able to get the request state that means that
202   // this track either isn't in the library, or its in the library but
203   // being launched in a manner where we don't yet have the library
204   // loaded.
205
206   CURL urlDesktopINI;
207   WString wszDesktopINI;
208   WString wszURLParams;
209
210   HRESULT hr = urlFolder.CopyTo( urlDesktopINI );
211   if (SUCCEEDED(hr))
212   {
213      hr = urlDesktopINI.PathAppend( L"desktop.ini" );
214   }
215   if (SUCCEEDED(hr))
216   {
217      hr = urlDesktopINI.Get( SF_FILEIO, wszDesktopINI );
218   }
219   if (SUCCEEDED(hr))
220   {
221      hr = wszURLParams.InitFixed( INTERNET_MAX_URL_LENGTH );
222   }
223   if (SUCCEEDED(hr))
224   {
225      wszURLParams.Clear();
226      (void)::GetPrivateProfileString( L".ShellClassInfo", L"MusicBuyUrl", L"", wszURLParams,
227   wszURLParams.GetFixedSize(), wszDesktopINI );
228      if (!wszURLParams.HasLength())
229      {
230         hr = E_INVALIDARG;
231      }
232   }
233   if (SUCCEEDED(hr))
234   {
235      WString wszFakeURL;
236      CURL urlParams;
237      WString wszVersion;
238
239      wszFakeURL.Init( L"http://www.foo.com?" );
240      wszFakeURL.Concat( wszURLParams );
```

```
241
242        hr = urlParams.Set( UI_DETECT, wszFakeURL );
243        if (SUCCEEDED(hr))
244        {
245            hr = urlParams.GetParameter( L"Version", wszVersion );
246        }
247        if (SUCCEEDED(hr))
248        {
249            long nVersion = wszVersion.CopyToLong();
250            if (nVersion >= 9)
251            {
252                //
253                // OK, we got it...  the 9.0 player wrote this file.
254                // This guarantees that the file was for a different
255                // piece of media (or we would have matched a {GUID} art
256                // from above.  Therefore, we ignore this folder.jpg
257                //
258
259                // Update:  Turns out we occasionally write desktop.ini when
260                //          we find metadata, but don't download album art for
261                //          a particular file.  This makes throwing things out
262                //          at this point somewhat problematic as we might have
263                //          matched data, but not downloaded art.
264                //
265                //          So we're now going to do a further check to see if
266                //          there is any GUID art in the folder at all... if there
267                //          is, then we can be confident that we're doing the right
268                //          thing by throwing it out.
269
270                WString wszFolder;
271                if (   SUCCEEDED( urlFolder.Get( SF_FILEIO, wszFolder ) ) &&
272                       wszFolder.HasLength()                              )
273                {
274                    if (wszFolder[ wszFolder.Length() - 1 ] != L'\\')
275                    {
276                        wszFolder.Concat( L'\\' );
277                    }
278                    wszFolder.Concat( g_kwszAlbumArtPrefix );
279                    wszFolder.Concat( L'*' );
280                    wszFolder.Concat( g_kwszArtSuffixLarge );
281
282                    WIN32_FIND_DATA FindData    = {0};
283                    bool            fFoundArt   = false;
284                    HANDLE          hFind       = ::FindFirstFile( wszFolder, &FindData );
285                    if (hFind != INVALID_HANDLE_VALUE)
286                    {
287                        fFoundArt = true;
288                        ::FindClose( hFind );
```

```
289                    hFind = INVALID_HANDLE_VALUE;
290                }
291
292            if (fFoundArt)
293            {
294                return false;
295            }
296        }
297      }
298    }
299    }
300
301    // Use the Helix art.
302
303    return true;
304 }
305
306
307
308 //+------------------------------------------------------------------------
309 //+------------------------------------------------------------------------
310
311
312 #define SHGVSPB_PERUSER          0x00000001 // must have one of PERUSER or ALLUSERS
313 #define SHGVSPB_PERFOLDER         0x00000004 // must have one of PERFOLDER ALLFOLDERS or
314 INHERIT
315 #define SHGVSPB_FOLDER           (SHGVSPB_PERUSER | SHGVSPB_PERFOLDER)
316
317 //*****************************************************************************/
318 typedef HRESULT (WINAPI* SHGETVIEWSTATEPROPERTYBAG)(LPCITEMIDLIST, LPCWSTR, DWORD, REFIID,
319 void**);
320 typedef HRESULT (WINAPI* SHPARSEDISPLAYNAME)(PCWSTR, IBindCtx *, LPITEMIDLIST *, SFGAOF,
321 SFGAOF *);
322
323 #define ORDINAL_SHGetViewStatePropertyBag    515
324
325 BOOL GetCustomAlbumArt( WCHAR * wszMusicDirName, VARIANT *pvtitem )
326 {
327    HRESULT             hr = S_OK;
328    HINSTANCE               hDLLShlObj = NULL;
329    HINSTANCE            hDLL = NULL;
330    SHPARSEDISPLAYNAME       pfnSHParseDisplayName = NULL;
331    SHGETVIEWSTATEPROPERTYBAG pfnSHGetViewStatePropertyBag = NULL;
332    LPITEMIDLIST          pidl = NULL;
333    CComPtr<IPropertyBag>    spPropertyBag;
334    CComVariant          varCustomFolder;
335
336    if ( IsWhistlerOrBetter() )
```

```
337      {
338         hDLLShlObj = LoadLibrary(L"SHELL32.DLL");
339         if (hDLLShlObj == NULL)
340         {
341            hr = HRESULT_FROM_WIN32( SAFE_GETLASTERROR() );
342         }
343      }
344      else
345      {
346         hr = E_NOTIMPL;
347      }
348
349      if( SUCCEEDED( hr ) )
350      {
351         pfnSHParseDisplayName = (SHPARSEDISPLAYNAME) GetProcAddress(hDLLShlObj,
352   "SHParseDisplayName");
353         if ( NULL == pfnSHParseDisplayName )
354         {
355            hr = HRESULT_FROM_WIN32( SAFE_GETLASTERROR() );
356         }
357      }
358
359      if( SUCCEEDED( hr ) )
360      {
361         hr = pfnSHParseDisplayName( wszMusicDirName, NULL, &pidl, 0, NULL );
362      }
363
364      if( SUCCEEDED( hr ) )
365      {
366         hDLL = LoadLibrary(L"SHLWAPI.DLL");
367         if (hDLL == NULL)
368         {
369            hr = HRESULT_FROM_WIN32( SAFE_GETLASTERROR() );
370         }
371      }
372
373      if( SUCCEEDED( hr ) )
374      {
375         pfnSHGetViewStatePropertyBag = (SHGETVIEWSTATEPROPERTYBAG) GetProcAddress(hDLL,
376   (LPCSTR) ORDINAL_SHGetViewStatePropertyBag);
377         if ( NULL == pfnSHGetViewStatePropertyBag )
378         {
379            hr = HRESULT_FROM_WIN32( SAFE_GETLASTERROR() );
380         }
381      }
382
383      if( SUCCEEDED( hr ) )
384      {
```

Application of Plastina et al.
Serial No. 10/622,767

Exhibit E
MS 303015.01 (5052)

```
385        hr = pfnSHGetViewStatePropertyBag( pidl, L"Shell", SHGVSPB_FOLDER, __uuidof(IPropertyBag),
386    (void **) &spPropertyBag );
387        }
388
389      if( SUCCEEDED( hr ) )
390      {
391        hr = spPropertyBag->Read( L"Logo", &varCustomFolder, NULL );
392      }
393
394      if( SUCCEEDED( hr ) )
395      {
396        if ( VT_BSTR == V_VT(&varCustomFolder) && ( NULL != varCustomFolder.bstrVal ) && ( 0 != wcslen(
397    varCustomFolder.bstrVal ) ))
398        {
399          if ( WMPHelper::DoesFileExist( varCustomFolder.bstrVal ) )
400          {
401            hr = varCustomFolder.Detach(pvtitem);
402          }
403        }
404        else
405        {
406          hr = E_INVALIDARG; // No art we like, invalid arg
407        }
408      }
409
410      if( NULL != pidl )
411      {
412        CComPtr<IMalloc> spMalloc;
413
414        if( SUCCEEDED( SHGetMalloc( &spMalloc ) )&& spMalloc )
415        {
416          spMalloc->Free( pidl );
417        }
418      }
419
420      if( NULL != hDLLShlObj )
421      {
422        FreeLibrary( hDLLShlObj );
423      }
424
425      if( NULL != hDLL )
426      {
427        FreeLibrary( hDLL );
428      }
429
430      return ( SUCCEEDED( hr ) );
431    }
432
```

```
433
434    //+------------------------------------------------------------------------
435    //
436    //  Function:   GetImageHandleFromFile
437    //
438    //  Opens the specified file using the WM Metadata Editor and extracts any
439    //  attached image data from the file into a handle.
440    //
441    //------------------------------------------------------------------------
442
443    HRESULT
444    GetImageHandleFromFile(const WCHAR *pszFilePath, HGLOBAL *phImage, DWORD *pdwImageSize)
445    {
446       CComPtr<IWMMetadataEditor>  spEditor;
447       CComPtr<IWMHeaderInfo>     spHeaderInfo;
448       HGLOBAL              hImageToReturn = NULL;
449       DWORD                dwImageLenToReturn = 0;
450       HRESULT              hr;
451
452       // open the file using the metadata editor
453       hr = WMCreateEditor( &spEditor );
454       if (FAILED(hr))
455          goto FAILURE;
456
457       hr = spEditor->Open(pszFilePath);
458       if (FAILED(hr))
459          goto FAILURE;
460
461       // get the header info interface
462       hr = spEditor->QueryInterface(&spHeaderInfo);
463       if (FAILED(hr))
464          goto FAILURE;
465
466       // determine how big the data is.
467       WORD wStream = 0;
468       WMT_ATTR_DATATYPE datatype = (WMT_ATTR_DATATYPE) -1;
469       WORD wDataLength = 0;
470
471       hr = spHeaderInfo->GetAttributeByName(&wStream, g_wszWMPicture, &datatype, NULL,
472    &wDataLength );
473
474       if ((FAILED(hr)) || (0 == wDataLength) || (WMT_TYPE_BINARY != datatype))
475       {
476          goto FAILURE;
477       }
478
479       // allocate space for the data we are about to read
480       BYTE *pbData = new BYTE[wDataLength];
```

```
481      if (NULL == pbData)
482      {
483         hr = E_OUTOFMEMORY;
484         goto FAILURE;
485      }
486
487      HGLOBAL hImageHandle = NULL;
488
489      // read the data
490      hr = spHeaderInfo->GetAttributeByName(&wStream, g_wszWMPicture, &datatype, pbData,
491  &wDataLength );
492
493      if ((SUCCEEDED(hr)) && (0 != wDataLength) && (WMT_TYPE_BINARY == datatype))
494      {
495         WM_PICTURE  *pPicture = (WM_PICTURE *) pbData;
496
497         if ((pPicture->dwDataLen) && (pPicture->pbData))
498         {
499            // create memory handle to hold image data
500            hImageHandle = ::GlobalAlloc( GMEM_MOVEABLE | GMEM_ZEROINIT, pPicture->dwDataLen );
501
502            if (hImageHandle)
503            {
504               void *pImageBuffer = ::GlobalLock( hImageHandle );
505               if (pImageBuffer)
506               {
507                  // copy image data to handle
508                  memcpy(pImageBuffer, pPicture->pbData, pPicture->dwDataLen);
509                  ::GlobalUnlock(hImageHandle);
510
511                  hImageToReturn = hImageHandle;
512                  dwImageLenToReturn = pPicture->dwDataLen;
513                  hImageHandle = NULL; // don't delete this below
514               }
515
516               if (hImageHandle)
517               {
518                  ::GlobalFree( hImageHandle );
519               }
520            }
521         }
522      }
523
524      delete [] pbData;
525
526      if (NULL == hImageToReturn)
527      {
528         hr = ASF_E_NOTFOUND;
```

```
529        goto FAILURE;
530      }
531
532      // return image handle
533      *phImage = hImageToReturn;
534      *pdwImageSize = dwImageLenToReturn;
535
536      return S_OK;
537
538  FAILURE:
539      if (hImageToReturn)
540        ::GlobalFree(hImageToReturn);
541
542      return hr;
543  }
544
545  //+----------------------------------------------------------------------
546  //
547  //  Function:   GetImageHandleFromStream
548  //
549  //  Extracts any attached image data from the stream into a handle.
550  //
551  //----------------------------------------------------------------------
552
553  HRESULT
554  GetImageHandleFromStream(IWMHeaderInfo *pHeaderInfo, HGLOBAL *phImage, DWORD
555  *pdwImageSize)
556  {
557      HRESULT              hr;
558      CComPtr<IWMHeaderInfo3> spHeaderInfo3;
559
560      // need a IWMHeaderInfo3
561      hr = SafeQueryInterface(pHeaderInfo, &spHeaderInfo3);
562      if ((FAILED(hr)) || (!spHeaderInfo3))
563      {
564        return E_NOINTERFACE;
565      }
566
567      // determine how many pictures are in our file
568      WORD wLanguageIndex = 0;
569      WORD wNumIndices = 0;
570
571      hr = spHeaderInfo3->GetAttributeIndices(0, g_wszWMPicture, &wLanguageIndex, NULL,
572  &wNumIndices);
573      if ((FAILED(hr)) || (0 == wNumIndices))
574      {
575        return E_FAIL;
576      }
```

```
577
578      // read the indices
579      WORD *pwIndices = pwIndices = new WORD[wNumIndices];
580      if (NULL == pwIndices)
581      {
582         return E_OUTOFMEMORY;
583      }
584
585      wLanguageIndex = 0;
586      hr = spHeaderInfo3->GetAttributeIndices(0, g_wszWMPicture, &wLanguageIndex, pwIndices,
587   &wNumIndices);
588      if (FAILED(hr))
589      {
590         delete [] pwIndices;
591
592         return hr;
593      }
594
595      HGLOBAL hImageToReturn = NULL;
596      DWORD dwImageLenToReturn = 0;
597
598      // loop over all pictures found
599      for (WORD wIndex = 0; wIndex < wNumIndices; wIndex++)
600      {
601         // determine how big the data is.
602         WORD wNameLen = 0;
603         WMT_ATTR_DATATYPE datatype = (WMT_ATTR_DATATYPE) -1;
604         DWORD dwDataLength = 0;
605         wLanguageIndex = 0;
606
607         hr = spHeaderInfo3->GetAttributeByIndexEx(0, pwIndices[wIndex], NULL, &wNameLen,
608   &datatype,
609                                   &wLanguageIndex, NULL, &dwDataLength);
610
611         if ((FAILED(hr)) || (0 == dwDataLength) || (WMT_TYPE_BINARY != datatype))
612         {
613            continue;
614         }
615
616         // allocate space for the data we are about to read
617         BYTE *pbData = new BYTE[dwDataLength];
618         if (NULL == pbData)
619         {
620            continue;
621         }
622
623         HGLOBAL hImageHandle = NULL;
624         wLanguageIndex = 0;
```

```
625
626         // read the data
627         hr = spHeaderInfo3->GetAttributeByIndexEx(0, pwIndices[wIndex], NULL, &wNameLen,
628    &datatype,
629                                    &wLanguageIndex, pbData, &dwDataLength);
630
631         if ((SUCCEEDED(hr)) && (0 != dwDataLength) && (WMT_TYPE_BINARY == datatype))
632         {
633            WM_PICTURE  *pPicture = (WM_PICTURE *) pbData;
634
635            if ((pPicture->dwDataLen) && (pPicture->pbData))
636            {
637               // create memory handle to hold image data
638               hImageHandle = ::GlobalAlloc( GMEM_MOVEABLE | GMEM_ZEROINIT, pPicture-
639    >dwDataLen );
640
641               if (hImageHandle)
642               {
643                  void *pImageBuffer = ::GlobalLock( hImageHandle );
644                  if (pImageBuffer)
645                  {
646                     // copy image data to handle
647                     memcpy(pImageBuffer, pPicture->pbData, pPicture->dwDataLen);
648                     ::GlobalUnlock(hImageHandle);
649
650                     // hang onto first image in case we don't find any others
651                     if (NULL == hImageToReturn)
652                     {
653                        hImageToReturn = hImageHandle;
654                        dwImageLenToReturn = pPicture->dwDataLen;
655                        hImageHandle = NULL; // don't delete this below
656                     }
657
658                     // otherwise see if this is the cover art image and hang onto that
659                     else if (0x0003 == pPicture->bPictureType)  // value 0x0003 defined as cover art by the
660    ID3 spec
661                     {
662                        ::GlobalFree(hImageToReturn);
663                        hImageToReturn = hImageHandle;
664                        dwImageLenToReturn = pPicture->dwDataLen;
665                        hImageHandle = NULL; // don't delete this below
666                     }
667                  }
668
669                  if (hImageHandle)
670                  {
671                     ::GlobalFree( hImageHandle );
672                  }
```

```
673              }
674            }
675          }
676
677        delete [] pbData;
678      }
679
680      delete [] pwIndices;
681
682      if (hImageToReturn)
683      {
684         *phImage = hImageToReturn;
685         *pdwImageSize = dwImageLenToReturn;
686         return S_OK;
687      }
688
689      return E_FAIL;
690  }
691
692  //+------------------------------------------------------------------------
693  //
694  //  Function:   CheckMediaForImages
695  //
696  //  Returns whether the stream has any attached images. This function is used
697  //  to quickly determine if further attached image processing is necessary.
698  //------------------------------------------------------------------------
699
700  HRESULT
701  CheckMediaForImages(IWMHeaderInfo* pHeaderInfo, BOOL *pfHasImages)
702  {
703      WORD            wStream = 0;
704      WORD            wLanguageIndex = 0;
705      WORD            wNumIndices = 0;
706      WMT_ATTR_DATATYPE   Type = (WMT_ATTR_DATATYPE) -1;
707      CComPtr<IWMHeaderInfo3> spHeaderInfo3;
708      HRESULT         hr = S_OK;
709
710      //
711      // In Corona the easiest way to check for images is to
712      // call GetAttributeIndices (WM/Picture).  That will cover all
713      // images in ASF and MP3 files except for Music Match
714      // "ID3" attribute images in WMA files.  To detect those
715      // you need to call GetAttributeByName (WM/Picture).  However,
716      // that only works on the editor and right here we're using the
717      // reader.  So to get a rough answer, we look for the "ID3"
718      // attribute that the MusicMatch images are contained in.  If this
719      // attribute is present, then there might be images in it and we will then take
720      // the perf hit later to open an instance of WM Editor to check for them.
```

```
721      //
722
723      // need a IWMHeaderInfo3
724      hr = SafeQueryInterface(pHeaderInfo, &spHeaderInfo3);
725      if ((FAILED(hr)) || (!spHeaderInfo3))
726      {
727         return E_NOINTERFACE;
728      }
729
730      //
731      // Look for images the regular way
732      // This will check the SDKs attribute cache so it will be quick
733      //
734      hr = spHeaderInfo3->GetAttributeIndices(0, g_wszWMPicture, &wLanguageIndex, NULL,
735  &wNumIndices);
736      if (SUCCEEDED(hr) && (0 < wNumIndices))
737      {
738         //
739         // It has at least one regular image
740         //
741         *pfHasImages = TRUE;
742         return S_OK;
743      }
744
745      //
746      // We couldn't find any images the regular way
747      //
748      // MusicMatch stores ID3 tags in a WMA file by storing one
749      // binary attribute called "ID3" in the WMA header. This binary
750      // attribute contains all the ID3 tags MusicMatch had defined
751      // for this file, including the "APIC" tag for embedded images.
752      // The WM Editor knows how to walk this binary attribute and
753      // extract all the ID3 tag information from it.
754      //
755      // Unfortunately the WM Reader does not parse this attribute and
756      // will just skip all the tags in it. This means the code above to
757      // check for the presence of images in a file will always fail
758      // for WMA files.
759      //
760      // This will be relatively quick also.  It check the regular cache first,
761      // then check the embedded MusicMatch attribute cache
762      //
763      WORD cbLength = 0;
764      hr = pHeaderInfo->GetAttributeByName( &wStream, L"ID3", &Type, NULL, &cbLength );
765      if (SUCCEEDED(hr))
766      {
767         //
768         // This isn't a guarantee that it has images, but there's a good chance
```

```
769        //
770          *pfHasImages = TRUE;
771          return S_OK;
772        }
773
774      *pfHasImages = FALSE;
775
776      return S_OK;
777    }
778
779    //+-------------------------------------------------------------------------
780    //
781    // Function:   CWMPMedia::GetAlbumArtFromMedia
782    //
783    // Gets album art embedded in the media.
784    //
785    // Album art images are only extraced from the media if there are no album art
786    // urls already associated with the media. Typically embedded album art images will
787    // come from the ID3 tag "APIC".
788    //
789    //-------------------------------------------------------------------------
790
791    HRESULT
792    CWMPMedia::GetAlbumArtFromMedia(IWMHeaderInfo* pHeaderInfo)
793    {
794        HRESULT    hr;
795        HGLOBAL     hImage = NULL;
796        DWORD       dwImageSize = 0;
797        IStream*    pStream = NULL;
798
799        // do a cheap check using the WM Reader to see if there are any image
800        // tags in the stream. This is so we avoid opening a new WM Editor for
801        // every file just to check for album art.
802
803        BOOL fHasImages;
804
805        fHasImages = FALSE;
806        hr = CheckMediaForImages(pHeaderInfo, &fHasImages);
807        if(FAILED(hr))
808            goto FAILURE;
809
810        if (!fHasImages)
811        {
812            hr = ASF_E_NOTFOUND;
813            goto FAILURE;
814        }
815
816        // try to get image from stream
```

```
817       hr = GetImageHandleFromStream(pHeaderInfo, &hImage, &dwImageSize);
818       if (FAILED(hr))
819       {
820          // could not get image from stream, so open the file using the
821          // WM Metadata Editor and get the image from there
822
823          // make sure file is local
824          VARIANT_BOOL vtIsLocal = VARIANT_FALSE;
825
826          hr = IsLocal(&vtIsLocal);
827          if ( (SUCCEEDED(hr)) && (VARIANT_TRUE == vtIsLocal) )
828          {
829             // get file path
830             WBSTRString bstrFullFilename;
831
832             hr = get_sourceURL(&bstrFullFilename);
833             if ( (SUCCEEDED(hr)) && (bstrFullFilename.HasLength()))
834             {
835                hr = GetImageHandleFromFile(bstrFullFilename, &hImage, &dwImageSize);
836                if (FAILED(hr))
837                   goto FAILURE;
838             }
839          }
840          else
841          {
842             goto FAILURE;
843          }
844       }
845
846       NSASSERT(hImage);
847
848       // create a stream out of the image data
849       hr = ::CreateStreamOnHGlobal(hImage, TRUE, &pStream);
850       if(FAILED(hr))
851          goto FAILURE;
852
853       NSASSERT(pStream);
854
855       // the IStream now owns this memory, so don't dispose it below
856       hImage = NULL;
857
858       ULARGE_INTEGER uliSize;
859       uliSize.LowPart = dwImageSize;
860       uliSize.HighPart = 0;
861
862       hr = pStream->SetSize( uliSize );
863       if(FAILED(hr))
864          goto FAILURE;
```

Application of Plastina et al.
Serial No. 10/622,767

Exhibit E
MS 303015.01 (5052)

```
865
866        // make this stream both the small and large image
867        VARIANT var;
868        VariantInit(&var);
869        var.vt = VT_UNKNOWN;
870        var.punkVal = pStream;
871
872        hr = SetReservedItem(kmriidSmallAlbumArtImage, var, VARIANT_TRUE);
873        if(FAILED(hr))
874           goto FAILURE;
875
876        hr = SetReservedItem(kmriidLargeAlbumArtImage, var, VARIANT_TRUE);
877        if(FAILED(hr))
878           goto FAILURE;
879
880        // the SetReservedItem function did an AddRef on the IStream, so we can release our ref now
881        pStream->Release();
882
883        return S_OK;
884
885 FAILURE:
886     if (pStream)
887        pStream->Release();
888
889     if (hImage)
890        ::GlobalFree(hImage);
891
892     return hr;
893 }
894
895
896
897
```